Parallelization and Auto-scheduling of Data Access Queries in ML Workloads

Pawel Bratek[1(✉)], Lukasz Szustak[1], and Jaroslaw Zola[2]

[1] Czestochowa University of Technology, Dabrowskiego 69, 42-201 Czestochowa, Poland
{pawel.bratek,lukasz.szustak}@pcz.pl
[2] University at Buffalo, Buffalo, NY 14260, USA
jzola@buffalo.edu

Abstract. We propose an auto-scheduling mechanism to execute counting queries in machine learning applications. Our approach improves the runtime efficiency of query streams by selecting, in the on-line manner, the optimal execution strategy for each query. We also discuss how to scale up counting queries in multi-threaded applications.

Keywords: Data access queries · Auto-scheduling · Machine learning · SABNAtk

1 Introduction and Problem Formulation

Counting data records with instances that support some specific configuration of the selected variables is one of the basic operations used by machine learning algorithms. The problem manifests itself each time a probability distribution has to be estimated, and spans applications ranging from Bayesian networks learning through association rule mining and classification [2,4] all the way to deep learning [6] and information retrieval [5].

Counting is typically viewed as a black-box procedure, and implemented using simple and not necessarily efficient strategies, e.g., contingency tables. At the same time, in many applications it accounts for over 90% of the total execution time [1]. Consequently, improving performance of counting can directly translate into better performance of these applications. The current specialized approaches based on data indexing, such as ADtrees [3], have limited applicability due to the significant preprocessing and memory overheads. Recently, Karan et al. [1] proposed SABNAtk, a new strategy in which counting queries and their context of execution are abstracted such that the counts can be aggregated as a stream, irrespective of the user-defined downstream processing.

Consider a set of n categorical random variables $\mathcal{X} = \{X_1, X_2, \ldots, X_n\}$, where the domain of variable X_i is represented by r_i states $[x_{i1}, \ldots, x_{ir_i}]$. Let $D = [D_1, D_2, ..., D_n]$ be a complete database of instances of $\mathcal{X}$, where D_i, $|D_i| = m$, records observed states of X_i. Given the set of input variables

R. Chaves et al. (Eds.): Euro-Par 2021, LNCS 13098, pp. 525–529, 2022.
https://doi.org/10.1007/978-3-031-06156-1_43

$\{X_i, X_j, \ldots\} \subseteq \mathcal{X}$ represented by database D, the counting query $\textsc{Count}((X_i = x_i) \wedge (X_j = x_j) \wedge \ldots)$ returns the size of the support in D for the specific assignment $[x_i, x_j, \ldots]$ of variables $\{X_i, X_j, \ldots\}$. For example, the result of executing query $\textsc{Count}((X_1 = 3) \wedge (X_2 = 1) \wedge (X_3 = 2))$ over database D shown in Fig. 1 is 2, because there are two instances matching the query condition. Simple counting query generalizes to a set of queries over the same set of variables $\mathrm{COUNT}(X_i, X_j, \ldots)$, in which we want to retrieve all non-zero answers to queries $\textsc{Count}((X_i = x_i) \wedge (X_j = x_j) \wedge \ldots)$ for all possible assignments of query variables. We will say that individual queries within COUNT query share context. For example, the answer to query $\mathrm{COUNT}(X_1, X_3)$ would return the following list of counts: $[((1,2),2), ((2,1),2), ((3,2),3)]$, where each entry is in the form $((x_i, x_j), \textsc{Count}((X_1 = x_i) \wedge (X_3 = x_j)))$.

The starting point for our work are counting strategies proposed in [1], and implemented in the open-source C++17 library SABNAtk. In SABNAtk, the counting queries COUNT can be answered using one of three strategies: 1) simple contingency table (`CT`), in which contingency table over all possible states of query variables is constructed, 2) bitmap counter (`BC`) in which input data is represented via bitmaps and counting is reduced to bitmap intersecting and bit counting, and 3) radix counter (`RC`) in which counting is based on columnar data partitioning similar to radix sorting. Which of the strategies is the best depends on many factors (e.g., query variables, data complexity, etc.) and hence it is not possible to state *a-priori* that one strategy dominates the others (see Fig. 2). In this work, we focus on scaling up SABNAtk in multi-threaded applications. Specifically, we seek to introduce a new auto-scheduling mechanism that learns online strategy to select optimal query processing counter.

	D_1	D_2	D_3
1	2	2	1
2	3	1	2
3	1	3	2
4	1	2	2
5	2	1	1
6	1	1	2
7	3	1	2
8	3	2	2

Fig. 1. Toy example of database D with three variables.

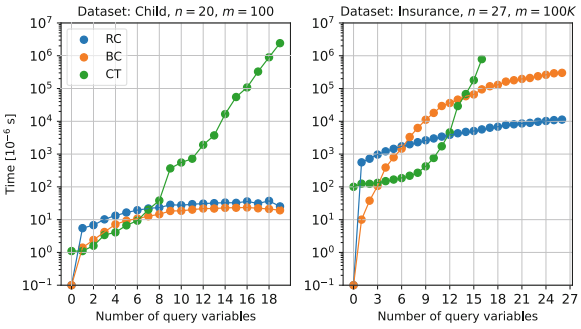

Fig. 2. Execution time of counting queries by different query strategies depending on the number of query variables (shorter is better). Each point represents an average time of ten randomly generated queries

2 Proposed Approach and Preliminary Results

To enable multi-threaded execution, we focus on two main questions: 1) how to efficiently execute any individual query, and 2) how to deal with a batch of queries generated concurrently by multiple threads? The strategies implemented within SABNAtk are stateless. Consequently, the simplest approach is to execute a query within the thread that issued it. However, in the real-world applications, it is common that consecutive queries share some of the query variables (i.e., context). Hence we propose to introduce a queuing and query rewriting mechanism to mitigate redundant data accesses.

To address the first question, we first experimentally assess and then theoretically characterize performance of each of SABNAtk's counting strategies. Figure 2 shows one representative example of execution profile. From the figure, it follows that the choice of the optimal strategy is non-trivial. At the same time, choosing the right counter offers significant reduction in the query execution time.

Our key idea is to pose the problem of selecting the optimal query execution strategy as an online regression problem. To this end, we first analytically derive the asymptotic average complexity of each counter as a function of the query size and the query complexity. The regression function is fit on-the-fly concurrently with serving the queries. Initially, the choice of strategy is random to mitigate overfitting, and as the execution proceeds, it becomes guided by our trained regression function.

Consider a single query $\mathrm{COUNT}(X_1, X_2, \ldots, X_N)$ of size N. Additionally, let $q = \prod_{i=1}^{N} r_i$ be the product of arity of query variables. The cost of the `CT` strategy is given by $N \cdot m + q$ in both worst and average case. The average complexity of `BC` depends directly on the input data and hence is difficult to characterize exactly. Therefore, we make a simplifying assumption that each variable $X_i \in \mathcal{X}$ is derived from a multinomial distribution with K equiprobable states $[x_{i1}, ..., x_{i_K}]$. Then from the Bernoulli scheme and the properties of expectation we can derive the cost as $m \times \sum_{L=0}^{N} K^L \cdot (1 - (1 - \frac{1}{K^L})^m)$. Finally, the cost of `RC` is asymptotically linear and amounts to $N \cdot m$. The derived average complexities allow us to define the following functions as query execution cost predictors:

$$CT(N, K, m, \boldsymbol{\beta}) = \beta_1 \cdot N \cdot m + \beta_2 \cdot K^{\beta_3 \cdot N} + \beta_4$$

$$BC(N, K, m, \boldsymbol{\beta}) = \begin{cases} \beta_1 \cdot m \cdot \dfrac{(K^{\beta_2 \cdot N + 1} - 1)}{(K - 1)} & N \leq N_0 \\ BC(N_0, K, m, \beta_1, \beta_2) + \beta_3 \cdot (N - N_0) \cdot m + \beta_4 & N > N_0, \end{cases}$$

$$RC(N, m, \boldsymbol{\beta}) = \beta_1 \cdot N \cdot m + \beta_2,$$

where N_0 is a number satisfying the condition: $K^{N_0} < m \wedge K^{N_0+1} > m$. The parameter K follows from the assumption about input data, and in practice, can be replaced by the arithmetic mean of the arity of variables included in the query (or any other meaningful statistics, e.g., median, etc.).

We use defined functions to build online regression model $Y = f(\boldsymbol{X}, \boldsymbol{\beta}) + \epsilon$ for each counter. Our approach assumes a stream of incoming queries that initially

are performed by randomly selected counters. We count cycles needed for their executions using Performance Application Programming Interface (PAPI) and apply them to the regression model as observations of the dependent variable Y. For a given query, we choose a counter with the smallest estimated cost of execution. We update the values of vector $\boldsymbol{\beta}$ after each query realization what results in better knowledge about the efficiency of counters depending on the query complexity.

In Table 1 we outline our preliminary results. Here the improvement factor shows how the auto-scheduling mechanism improves the overall performance. To obtain the baseline, we executed 1000 randomly generated queries for each of the presented configurations, and collected the total realization time of these queries using a randomly selected counter for each of them. Then we processed exactly the same query stream using our proposed auto-scheduling mechanism. The improvement factor is the ratio of the two runtimes. As shown in Table 1, our strategy offers improvement from 10.74× to 778.21× depending on the input data. These are very significant improvements considering that in real-world scenarios average ML application has to handle billions of queries.

Table 1. Improvement factor with respect to random strategy

Dataset	n	Improvement factor		
		$m = 1K$	$m = 10K$	$m = 100K$
Child	20	778.21	87.95	22.62
Insurance	27	533.30	63.15	16.64
Mildew	35	282.58	19.17	26.69
Alarm	37	311.74	18.24	10.74
Barley	48	145.81	15.22	17.04

Currently, we work on query queuing and rewriting mechanism for multi-threaded environments. The problem is challenging as it requires careful choice of on-line strategies to decide when sufficient number of queries are queued to improve average query processing speed while maintaining acceptable latency.

Acknowledgments. This research was supported by the National Science Centre (Poland) under grant no. UMO-2017/26/D/ST6/00687.

References

1. Karan, S., Eichhorn, M., Hurlburt, B., Iraci, G., Zola, J.: Fast counting in machine learning applications. In: Uncertainty in Artificial Intelligence (2018)
2. Kohavi, R.: Scaling up the accuracy of Naive-Bayes classifiers: a decision-tree hybrid. In: International Conference on Knowledge Discovery and Data Mining, pp. 202–207 (1996)

3. Moore, A., Lee, M.: Cached sufficient statistics for efficient machine learning with large datasets. J. Artif. Intell. Res. **8**, 67–91 (1998)
4. Quinlan, J.: Bagging, boosting, and c4.5. In: AAAI Innovative Applications of Artificial Intelligence Conferences, pp. 725–730 (1996)
5. Ramos, J.: Using TF-IDF to determine word relevance in document queries. In: Instructional Conference on Machine Learning, pp. 133–142 (2003)
6. Salakhutdinov, R., Hinton, G.: Deep Boltzmann machines. In: International Conference on Artificial Intelligence and Statistics, pp. 448–455 (2009)